3D Electromagnetic Plasma Particle Simulations on a MIMD Parallel Computer

J. Wang and P. Liewer
Jet Propulsion Laboratory, California Institute of Technology
V. Decyk
University of California, Los Angeles

Abstract

A three-dimensional electromagnetic PIC code has been developed on the 512 node Intel Touchstone Delta MIMD parallel computer. This code uses a standard relativistic leapfrog scheme to push particles and a local finite- difference time-domain method to update the electromagnetic fields. The code is implemented using the General Concurrent PIC algorithm which uses a domain decomposition to divide the computation among the processors. The 3D simulation domain can be partitioned into 1-, 2-, or 3-dimensional subdomains. Particles must be exchanged between processors as they move among the subdomains. The Intel Delta allows one to use this code for very-large-scale simulations (i.e. over 10^8 particles and 10^6 grid cells). The parallel efficiency of this code is measured, and the overall code performance on the Delta is compared with that on Cray supercomputers. It is shown that our code runs with a high parallel efficiency of $\geq$ 95% for large size problems. The particle push time achieved is 115 nsecs/particle/time step for 162 million particles on 512 nodes. Comparing with the performance on a single CPU Cray C90, this represents a factor of 58 speedup. It is also shown that the finite-difference method for field solve is significantly more efficient than transform methods on parallel computers. The field solve time is < 0.7% of total time for problems with 77 particles/cell, and it is < 3% even for problems with 7 particles/cell.

1 Introduction

Computer particle simulation has become a standard research tool for the study of non-linear kinetic problems in space and laboratory plasma physics research. A particle-in-cell (PIC) code simulates plasma. phenomena by modeling a plasma as hundreds of thousands of test particles and following the evolution of the orbits of individual test particles in the self-consistent electromagnetic field[1, 2]. Each time step in a PIC code consists of two major stages: the *particle push* to update the particle orbits and calculated the new charge and/or current density, and the *field solve* to update the electromagnetic fields, Since the particles can be located anywhere within the simulation domain but the macroscopic field quantities are defined only on discrete grid points, the particle push uses two interpolation steps to link the particle orbits and the field components: a "gather" step to interpolate fields from grid points to particle positions and a "scatter" step to deposit the charge/current of each particle to grid points.

While the particle simulation method allows one to study the plasma phenomena from the very fundamental level, the scope of the physics that can be resolved in a simulation study critically depends on the computational power. The computational time/cost and computer memory size restricts the time scale, spatial scale, and number of particles that can be used in a simulation. The cost of running three dimensional electromagnetic PIC calculations on existing sequential supercomputers limits the problems which can be addressed.

Recent advances in massively parallel supercomputers have provided computational possibilities that were previously not conceivable. For instance, the 512-processor Intel Touchstone Delta operated at Caltech by the Concurrent Supercomputing Consortium has 512 x 16 Mbytes = 8.19 gigabytes or 2,048 gigawords of memory and a peak speed of 512 x 80 single-precision Mflops = 40.96 single-precision gigaflops. The JPL T3D from Cray Research, if upgraded to 256 nodes with 8 Mwords per node, would also have 2.048 gigawords and a peak speed of 38 gigaflops.

Previously, one- and two-dimensional electrostatic and electromagnetic PIC codes have been implemented on MIMD parallel supercomputers using the General Concurrent PIC algorithm (GCPIC) which uses a domain decomposition to divide the computation among the processors[3, 4, 5]. This and other decomposition methods have also been studied by Walker [6] and Azari and Lee[7, 8]. More recently, the GCPIC implementation of the particle push portion of a three-dimensional electrostatic PIC code has been described and analyzed by Lyster et al[9]. In this paper we extend the previous work and describe the parallel implementation of both the particle and field stages of a three-dimensional electromagnetic PIC code. The objectives of "this study are to develop a three-dimensional electromagnetic PIC code for MIMD (multiple-instruction multiple-data) parallel supercomputers and to test the full potential of using parallel computers for very-large-scale particle simulations.

The code we developed uses a standard relativistic particle push and a local finite-difference time-domain solution to the full Maxwell's equations. This code is implemented on the 512-processor Intel Delta parallel computer using the GCPIC algorithm. The resulting parallel 3D EMPIC code has proven to be very efficient. For instance, for a test simulation using 162 million particles and 2.1 million grid cells on all 512 processors, we have achieved a parallel efficiency of 95% and a parallel push time of 115 nsecs/particle/time step. We also find that the finite-difference field solve is significantly more efficient than transform methods. For this test, run, the run time is dominated by the push time and the field solve time is < 0.7% of the total time. We have also run the same code on Cray supercomputers (the code was compiled using the Cray system's automatic vectorization and optimization, but no rewriting was clone to the code). Comparing with the performance on a single processor Cray C90, the run time we achieved on the Delta represents a factor of 58 speedup.

This paper is organized as follows: Section 2 discusses the algorithm used in our 3D EMPIC code and the parallel implementation; Section 3 analyzes the code performance through scaled size problems, fixed size problems, and a comparison of the performance on the Intel Delta with that on Cray supercomputers; and Section 4 contains a summary and conclusions.

2 A Parallel 3D Electromagnetic PIC Code

2.1 The Algorithm

An electromagnetic PIC code attempts to simulate plasma phenomena using only the fundamental physics laws, i.e. the Maxwell's equations for the macroscopic field and Newton's second law for individual particle trajectories:

$$\nabla \cdot \vec{E} = \rho \tag{1}$$

$$\nabla \cdot \vec{B} = 0 \tag{2}$$

$$\frac{\partial \vec{E}}{\partial t} = c \nabla \times \vec{B} - \vec{J} \tag{3}$$

$$\frac{\partial \vec{B}}{\partial t} = -c \nabla \times \vec{E} \tag{4}$$

$$\frac{d\gamma m\vec{V}}{dt} = \vec{F} = q(\vec{E} + \vec{V} \times \frac{\vec{B}}{c}), \quad \frac{d\vec{x}}{dt} = \vec{V} \tag{5}$$

where relativistic effects are included in eq(5) ($\gamma = 1/\sqrt{1 - V^2/c^2}$). Eqs(1) through (5) are the equations to be solved in our 3D electromagnetic PIC code.

in electromagnetic PIC codes, the field equations are often solved by transform methods such as fast Fourier transforms (FFT). However, transform methods are '(global" methods. in general, global methods are not very efficient for parallel computers because they involve a large amount of interprocessor communication which may eventually become the bottleneck. For a code to run efficiently in parallel, a method that updates the field purely from the local data is preferred. In our code, the electromagnetic field equations are solved using a charge-conserving finite-difference leapfrogging scheme, which was used by Sandia National Laboratories in the Quicksilver code [10, 11] and by Buneman et al in the Tristan code [12, 13]. This field solve scheme is described below.

From the Maxwell's equations, one notes that eq(1) will always be satisfied as long as the charge conservation condition

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \vec{J}$$

is satisfied. Hence, the electromagnetic field can be updated from only the two curl Maxwell's equations (3) and (4) if one can enforce rigorous charge conservation numerically. A rigorous charge conservation method for current deposit is described in detail in [12]. In this scheme, one obtains the current flux through every cell surface within a time step dt, $dt\vec{J}^{n+1/2}$, by counting the amount of charge carried across the cell surfaces by particles as they move from $\vec{x}^n$ to $\vec{x}^{n+1}$. Next, the electromagnetic field is updated *locally by* finite-difference leapfrogging in time:

$$\vec{E}^{n+1} - \vec{E}^n = dt\left[c \nabla \times \vec{B}^{n+1/2}\right] - dt\vec{J}^{n+1/2} \tag{6}$$

$$\vec{B}^{n+1/2} - \vec{B}^{n-1/2} = -dt\left[c \nabla \times \vec{E}^n\right] \tag{7}$$

where the superscripts $n + 1/2$ and $n + 1$ represent the time level. This scheme requires the use of a fully staggered grid mesh system in which $\vec{E}$ and $\vec{J}dt$ are defined at midpoints of cell-edges while the $\vec{B}$ components are defined at the midpoints of the ce]l-surfaces. The staggered grid mesh system, known in the computational electromagnetics community as the Yee lattice [14], is shown in Figure 1. It ensures that the change of B flux through a cell surface equals the negative circulation of E around that surface and the change of E flux through a cell surface (offset grid) equals the circulation of B around that surface minus the current through it.

In the code, the trajectory of each particle is integrated using a standard time-ccmtcring leapfrog scheme discussed in [1]. Let $\vec{u} = \gamma \vec{V}$, and the leapfrog scheme for eq(5) is written as

$$\vec{u}^{n+1/2} - \vec{u}^{n-1/2} = dt[\frac{q}{m}\vec{E}^n + \frac{\vec{u}^{n+1/2} , \vec{u}^{n-1/2}}{27''} \times \frac{q}{m}\frac{\vec{B}^n}{c}] \quad \textbf{(8)}$$

$$\vec{x}^{n+1} - \vec{x}^n = \frac{\vec{u}^{n+1/2}}{\gamma^{n+1/2}} dt \quad (9)$$

In eq(8), $\vec{E}$ and $\vec{B}$ are interpolated from the grids to the particle positions. See Ref.[1] for discussions on detailed steps for eq(8) and the centering of γ^n.

The basic algorithm for our electromagnetic PI C code is as follows:

1) Set the initial conditions of the particles and fields
(The initial conditions must satisfy the two divergence Maxwell
equations (1) and (2));

2) Particle Move:
a) interpolate the electromagnetic field on the particle position
to obtain the force on each particle (gather);
b) Update the particle velocity and position from eq(8) and (9);

3) Current Deposit (scatter):
Calculate the charge carried by particles across cell surfaces within the time step
to obtain the current flux $\vec{J}dt$ through each cell surface;

4) Field Update:
Solve the two curl Maxwell equations by finite- difference leapfrogging eq(6) and (i')
to update the electromagnetic field.

Steps 2) and 3) together form the particle push stage of the code.

2.2 Implementation on a MIMD Parallel Computer

Our 3D electromagnetic PIC code has been implemented on a MIMD parallel computer, the Intel Touchstone Delta at Caltech. The Intel Touchstone Delta system consists of an ensemble of nodes which are independent processors with their own memory connected as a two-dimensional mesh. There are 512 numerical nodes. Each node has a peak speed of 80 single-precision Mflops or 60 double-precision Mflops. The memory of each node is 16 Mbytes, of which 12 Mbytes are available for the user's code. Hence, the total available memory is an equivalent of 6.1 Gbytes on all 512 nodes.

The code is implemented using the General Concurrent PIC (GCPIC) algorithm developed by Liewer and Decyk[3]. The GCPIC algorithm is designed to make the most computationally intensive portion of a PIC code, the particle computation, run efficiently on MIMD parallel computers. nigh efficiency is achieved by minimizing interprocessor communication and balancing processor computational loads. In general, the GCPIC algorithm uses two spatial decompositions of the physical domain to divide the computation efficiently among parallel processors: a *primary decomposition* to optimize the parallel particle push computations (i.e., particle move and current deposit) and a *secondary* decomposition to optimize the parallel field computations (i.e., field update). In the primary decomposition, each processor is assigned a subdomain and all the particles and grid points in it. When a particle moves from one subdomain to another, it must be passed to the appropriate processors, which requires interprocessor communication. However, the primary decomposition is chosen so that both interpolations between the particles and the grids (gather/scatter) can be done *locally*, e.g., with no interprocessor communication. To ensure that the gather/scatter can be performed locally, each processor stores *guard cells*, e.g., neighboring grid points surrounding a processor's subdomain which belong to another processor's subdomain (Fig. 2). Interprocessor communication is necessary to exchange guard cell information. Depending on the method chosen for field update, the secondary decomposition can be either the same as or different from the primary decomposition. If the decompositions are distinct, additional interprocessor communication is necessary to move the grid data between the push and field stages at each time step [3, 15].

For our 3D EM PIC code with finite-difference field solve to have load balance, the primary decomposition subdomains should have roughly the same number of particles and the secondary decomposition subdomains should have the same number of grid points. When the grid is regular and the particle distribution is uniform, equal volume subdomains are optimum for both the push and field stages, 'l'bus, for this case, the primary and secondary decompositions are identical.

Fig. 2 illustrates a 2-dimensional subdomain. Each processors' subdomain is bounded in each dimension by

$$x_{left} \leq x < x_{right} \tag{1o}$$

The grid points within this subdomain are then

$$i_{left} \leq i \leq i_{right} \tag{11}$$

in each dimension, where $i_{left} = INT(x_{left}) + 1$ and $i_{right} = INT(x_{right})$ ($i_{left} = INT(x_{left})/$ $i_{right} = INT(x_{right}) - 1$ if x_{left}/x_{right} lies exactly o11 the grid p oint). Note that i_{left} and i_{right} denote the grid points on the global grid. In the code, the grid array indicies within a processor are based on the 'local" indexing. As illustrated in Fig. 3, in addition to the grid points from i_{left} to i_{right}, each processor also stores guard cells. If the number of guard cells on the left-side and right-side in each dimension are ng_l and ng_r respectively, then the total grid points stored in each processor are from $i_{left} - ng_l$ to $i_{right} + ng_r$ in each dimension. The number of the guard cells needed is determined by the particle weighting scheme as well as the algorithm to update the field. In this code, we use linear interpolation for particle weighting.

in our code, the computation domain can be partitioned into 1-, 2-, or 3-dimensional subdomains ("slabs", "rods", or "cubes") [9]. Fig. 3 shows the particles in typical 1-, 2-, and 3- di men sion al domain decompositions. The particles are colored according to processor. For a given problem, the optimal domain partition is chosen by considering many factors such as the problem size, the homogeneity of the problem, the number of processors that will be used, mac.rose.epic drifts, etc. Sine.c "productive" calculations are performed within a subdomain while interprocessor communications are through subdomain surface, communication cost correlates with the ratio of subdomain surface area to subdomain volume S/V. For the performance anal-ysis runs in this paper, we shall usc the cubic subdomain because it has the minimum surface to volume ratio S/V.

Figure 4 shows the flow chart of our parallel 3D electromagnetic PIC code. The main loop uses six major subroutines. *Particle Move, Current Deposit*, and *Field Update* for $\vec{E}$ and $\vec{B}$ (represented by the rounded blocks in Fig. 2) have been discussed in the last section. They are the essential computation blocks in a sequential EM PIC code. On a parallel computer-, each processor executes these operations independently using its own data arrays, The computations are linked together through message-passing and global communications. The code has three major message-passing subroutines: *Particle Trade*, *Guard Cell Summation*, and Guard *Cell Exchange* (represented by the five rectangular blocks), The global boundary conditions arc also imposed in these three subroutine to avoid additional loops over grid points and particles. Currently our code uses periodic boundary conditions. Hence, we have the processors logically connected periodically in the initial subdomain setup (e.g. the processors at right-most domain is connected to the processors at left-most domain). Therefore, periodic global boundary conditions are automatically imposed through communications between the left-most and right-most processors. Figure 4 also shows the two main stages for our parallel PIC code. The *particle push* consists of *Particle Move, Particle Trade*, *Current Deposit*, and *Guard Cell Summation*. The *field solve* consists of *Field Update* and *Guard Cell Exchange* for $\vec{E}$ and $\vec{B}$ fields.

Guard Cell Exchange is used to update the $\vec{E}$ and $\vec{B}$ fields at processor boundary cells and to impose the global periodic boundary conditions. The field grid and field guard cells are illustrated in Fig. 2. It is obvious from the finite difference scheme (eq(6) and eq(7)) that the grid points at i_{left} and i_{right} need the information at the grid points $i_{left} - 1$ and $i_{right} + 1$ respectively in order to be updated. Hence, the field update needs one guard cell surface on both the left and right side. The linear particle weighting scheme also requires one guard cell surface for interpolating the field to the particle position. Therefore, we take $ng_l = ng_r = 1$. In order to fill the guard cells with the updated $\vec{E}$ and $\vec{B}$, the $\vec{E}$ and $\vec{B}$ at i_{left} and i_{right} grid points need to be exchanged between the neighboring processors. The exchange of guard cells is done through a loop over the three dimenions x, y, and z with information exchanged separately in each of the three dimensions, Within the loop, the guard cells are exchanged in one dimension only. By performing the communication in each dimension separately, only two communication buffers and six communication calls are needed. (If all three dimensions were considered simultaneously, 26 communication buffers and calls would be needed[9].) The corner guard cells are also filled automatically after the three loops, The psuedo code for *guard cell exchange* subroutine is as follows:

```
For i=1, 3 dimensions do
```

```
    pack the fields at $i_{left}$ and $i_{right}$ surfaces into left-and right-going buffers
    send left- and right-going buffer to left- and right-neighbors
    receive from right- and left-neighbors
    unpack and fill the $i_{right}+1$ and $i_{left}+1$ surfaces
end do
```

Guard Cell Summation adds the currents deposited in guard cells to the proper cells in the neighboring processors and also imposes global boundary conditions. The current grids and guard cells are illustrated in Figure 2. When depositing current, those particles near a subdomain boundary will contribute currents to grid points within the processor's subdomain as well as the grid points which are owned by neighboring processors. Since the current is defined at the $n+1/2$ time level, the calculation of $J^{n+1/2}dt$ needs particle positions at both the $n+1$ and n time step ($\vec{x}^{n+1}$ and $\vec{x}^{n}$). To save storage as well particle communication, $\vec{x}^{n}$ is obtained by moving $\vec{x}^{n+1}$ backward in time in *current deposit*, which occurs after *particle trade*. Hence, for those particles which were traded at the $n+1$ time step, their $\vec{x}^{n}$s lie outside the subdomain of their current processor. Therefore, instead of one guard cell surface as required by the linear weighting scheme, one needs two guard cell surfaces on both side of the subdomain for current deposit, $ng_l = 2$ and $ng_r = 2$. (Note this is based on the usual constraint on time step in explicit PIC codes: $vdt <$ cell length). For the special case that x_{left} lies exactly at the i_{left} grid point, considering $vdt <$ cell length, only one guard cell surface cm the left side would be sufficient ($ng_l = 1$ and $ng_r = 2$), The guard cell currents need to be passed to the neighboring processors and added to the currents at the appropriate grid points of the neighboring processors. As in the *Guard Cell Exchange* described above, this guard cell communication is done separately for each dimension under a loop over dimensions x, y, and z. Guard cell contributions to the interior corner cells are automatically properly summed for after the three loops. The psuedo *guard cell summation* subroutine is as follows:

```
For i=1, 3 dimensions do
    pack currents in $i_{left}-ng_l$ to $i_{left}-1$ and $i_{right}+1$ to $i_{right}+ng_r$ surfaces
        into left- and right-going buffers respectively
    send left(right)-going buffer to left(right) neighbors
    receive buffers from left(right) neighbors and store in receive-left(right) buffers
    add the receive-right to currents at $i_{right} - ngr$ to $i_{right} - 1$ surfaces
    add the receive-left to currents at $i_{left} + 1$ to $i_{left} + ng_l$ surfaces
enddo
```

Note that for situations where the domain is not partitioned in a certain dimension, e.g., if one-dimensional "slab" or two-dimensional "rod" decompositions had been used instead of three-dimensional "cubes", then it is not necessary to do any interprocessor communication in this dimension. in this case, the code still loops over this dimension in the guard cell communications in *Guard* Cell *Exchange* and *Guard Cell Summation* routines in order to impose global periodic boundary conditions for the field and the current. However, no interprocessor communication occurs[9].

Particle Trade is used to trade particles between processors and impose global boundary conditions. After the Particle Move, each particles' new position is checked against the subdomains boundaries [eq. (10)]. If a particle is found out of bounds on the left (right), it is placed in a left (right)-going buffer[3]. When all particles in a processor have been checked, the buffers are passed to the neighboring processors, and at the same time, incoming particle buffers are received from the neighboring processors. The incoming particle buffer is then unpacked to fill in holes in the local particle array. The particle trade subroutine is a modification of the one described by Lyster et al[9]. For a detailed discussion of the method used to pack/unpack particle buffers, sce Liewer and Decyk[3]. The psuedo code for the *particle trade* subroutine is as follows:

```
For i= 1, 3-dimensions do
    for all particles do
        apply the periodic global boundary condition
            if particle position $< x_{left}$, pack into left-going buffer
            if particle position $\geq x_{right}$, pack into right-going buffer
        1   send left(right)-going buffer to left(right) neighbor
            receive buffers from left(right) neighbors and store in receive-left(right) buffers
            check particle positions in receive buffers:
                if a particle is at < $x_{left}$,
                    remove from receive-right and pack into left-goillg buffer
                if a particle is at ≥ $x_{right}$,
                    remove from rcccivc-left and pack into right-going buffer
            if there are particles need to be passed further, go to 1
            unpack receive-right/receive-left buffer into the local particle array
    end do particles
end do dimensions
```

Note that, as in the guard cell routines, if the domain is not partitioned in a certain dimension, after performing global boundary conditions, *Particle Trade* exits that dimension loop without pm-forming any interprocessor communication. The number of the particles in the receive-left(right) buffer is also checked before the buffer is unpacked to ensure that it would not overflow the local particle array. Finally, the step that checks particle positions in the receive-left(right) buffer is necessary for situations that particles may move more than one subdomain in a time step. For explicit PIC codes with static domain decomposition such as our code, particles will never travel more than one subdomain per time step due to numerical stability constraints on the time step (volt < cell length). However, particles may travel more than one subdomain per time step in PIC codes with dynamic load balance where the domain may be repartitioned every time step [] 5] or in implicit PIC codes where large dt may be used.

3 Performance Analysis

To analyze the performance of our parallel 3D EM PIC code, a simple test case was used: a relativistic. counter streaming electron beam instability. In this test case, two equal electron beams are set to counter stream in the x direction with drifting velocities $v_d = \pm 0.4c$. 'l'he electrons within each beam follow a Maxwellian distribution with thermal velocity $v_t = 0.05c$. The ions are considered as a fixed background. This counter streaming system generates the well-known two-stream instability. While the classical two-stream instability is an electrostatic instability, we find the instability generated here is electromagnetic in nature. Since the drifting speed is close to the speed of light, the unstable wave generated has comparable electric and magnetic fields. Fig. 5 shows a typical simulation result in the (x, z, v_x) phase space. This test run used 8 x 1 x 2 = 16 processors. The top panel shows the domain decomposition and the initial particle distribution with electrons colored by processor. The second and third panels show the electron distribution, colored by beam population, at $t = 0$ and $t =$ 47wljC1 respectively. The inter-mixing of the two beam populations caused by the instability is apparent in the third panel.

To evaluate the code performance, we have measured the total code time per time step loop T_{tot} as well as the times spent by each of the six major subroutines for a series of runs. Let us demote T_{move} , $T_{current}$, $T_{fldupdate}$, T_{trade}, T_{gcsm}, T_{gcfl} as the time spent by *particle move*, current deposit, *field update*, *particle trade*, *guard cell current summation*, and *field* guard cell *exchange* respectively. Since each processor runs the code with slightly different times, the times we measured are the maximum processor times on a parallel computer. Since the clock calls introduce synchronization, T_{tot} was measured with all the subroutine clocks turned off. There will be a small difference between the measured T_{tot} and the value of $T_{move} + T_{current} + T_{fldupdate} + T_{trade} + T_{gcsm} + T_{gcfl}$.

From the measured subroutine times, we define the particle push time (which includes the times on moving particles, depositing currents, applying boundary conditions, and related inter-processor communications as shown in Fig. 4):

$$T^{push} = T_{move} + T_{trade} + T_{current} + T_{gcsm} \tag{12}$$

and the field solve time (which includes the times on updating the E and B fields, applying boundary conditions, and related interprocessor communications as shown in Fig, 4):

$$T^{field} = T_{fldupdate} + T_{gcfl} \tag{13}$$

We also define the guard cell/boundary condition time as

$$T^{gc} = T_{gcsm} + T_{gcfl} \tag{14}$$

and the total communication /boundary condition time as

$$T^{cbc} = T_{trade} + T_{gcsm} + T_{gcfl} = T_{trade} + T^{gc} \tag{15}$$

If only one processor is used, $T^{cbc}(1)$ is simply the time spent on the global boundary conditions, When multiple processors are used, $T^{cbc}(N_p > 1)$ is the sum of the time spent by the code on communications and global boundary conditions. Let us denote T^{bc} to be the communication time spent for the purpose of global boundary conditions, S the area of the global domain surface, and S_{sum} the sum of the areas of all subdomain surface. The fraction of the global boundary condition time within T^{cbc} scales as

$$T^{bc}/T^{cbc} \sim S/S_{sum} \tag{16}$$

When the number of processors used is much larger then one, T^{cbc} "is dominated by the "pure" interprocessor communication and hence, it is a good measure of the parallel communication cost.

The performance of our parallel 3D electromagnetic PIC code is measured in three ways: 1) scaled problem size analysis; 2) fixed problem size analysis; and 3) comparison of the performance on the Intel Delta with that on 'single processor Cray supercomputers.

An important measure of the performance on a concurrent computer is the parallel efficiency ϵ which measures the effects of communication overhead and load imbalance[16]. If there were no communications involved and the processor loads were perfectly balanced, the parallel efficiency would be $\epsilon = 100\%$. In this paper we shall focus only on the effect due to communication overhead. The simulation runs used in this section all have near-perfect load balance because the particle distributions in these runs are nearly uniform. (Dynamic load balance for non-uniform particle distributions has been investigated in a 2D PIC code by Ferraro et al[15].)

3.1 Scaled Problem Size Analysis

We first study the parallel efficiency for very large simulations using a scaled problem size analysis. in a scaled problem size analysis, we keep the problem size on each individual processor fixed while increasing the total number of processors, The total problem size is then proportional to the number of processors used. The parallel efficiency in a scaled problem size analysis is defined as

$$\epsilon(N) = \frac{T_{tot}(1)}{T_{tot}(N)N} \tag{17}$$

where $T_{tot}(N)$ is the total loop time elapsed on a parallel computer using N nodes.

We consider two cases for scaled problem analysis. in both cases, cubic subdomains are used and the problems are scaled up evenly in the three dimensions. In the first case, S1, each node has 32 x 32 x 32 cells and 2.22 x 10^5 particles ($\sim$ 7 particles/cell). When S1 is loaded on all 512 nodes, the size of the total problem becomes 256 x 256 x 256 (16.8 million) cells and 114 million particles. In the second case, S2, each node has 16 x 16 x 16 cells and 3.16 x 10^5 particles ($\sim$ 77 particles/cell). The size of S2 on all 512 nodes is then 128 X 128 X 128 (2.1 million) cells and 162 million particles. We note that the memory size required to run S1 and S2 on each node are 10.4 Mbytes and 11.6 Mbytes respectively. Considering the memory available for calculation is 12 Mbytes per node on the Delta, S2 represents about the largest problem that one can fit onto the Delta system. When S2 is loaded to all 512 nodes of the Delta, the total memory size is an equivalent of 5.9 Gbytes.

The parallel efficiencies for S1 and S2 are shown in Fig. 6 (left axis) as a function of the number of processors N_p. The results show that a high parallel efficiency of $\epsilon \geq 95\%$ has been achieved for both S1 and S2. As mentioned before, since we have perfect load balance for the test runs, the efficiency is degraded only by interprocessor communications. Hence, the ratio of T^{cbc}/T_{tot} is also shown in Fig. 6 (right axis, note scale change). Wile.11 only 1 node is used, $T^{cbc}(1\,node)$ consists of only the global boundary condition time. The total boundary condition time is $T^{cbc}(1\,node) \simeq 0.025T_{tot}(1\,node)$ for S1 and $T^{cbc}(1\,node) \simeq 0.021T_{tot}(1\,node)$ for S2. When the number of processors is N_p >1, the major part of T^{cbc} "is for "pure" parallel communications. (For the scaled problems here, from eq(16), we have $T^{bc}/T^{cbc} \simeq 1/N_p^{1/3}$.) We find that T^{cbc} takes less then 5% of T_{tot} . Not surprisingly, when the number of processors used is much larger than 1, $\epsilon(N_p >> 1) \simeq 1 - T^{cbc}/T_{tot}$.

in Fig. 7, we plot the loop time T_{tot}, particle push time T^{push}, field solve time T^{field}, as well as the communication/boundary condition time for particle push $T_{trade} + T_{gcsm}$ and field solve T_{gcfl}. (Fig. 7a shows the times for S1 and Fig. 7b shows the times for S2.) We find T_{tot} is dominated by the particle push stage of the code (T^{push}), which stays almost constant as the number of processors is increased. In both cases, the field solve only represents a very small fraction of the total time: $T^{field}/T_{tot} \leq$ 3% for S1 and $T^{field}/T_{tot} \leq 0.66\%$ for **S2.** As a test, in some other simulations we have used $\sim$ 5 particles/cell, Even at such a low particle number/grid cell ratio, we find the field solve still takes $\leq$ 4% of the total time. As expected, the local finite difference field solve runs very fast in parallel PIC codes.

One of the most important measure of a PIC code's speed is the particle push time per particle per time step t^{push} or the total loop time per particle per time step t_{tot}. For S1 and S2, particle push times and the total loop time on the 512 node Delta are as follows:

$t^{push} \simeq 128$ ($t_{tot} \simeq 131$) nsecs/particle/time step for S1(1 14 million particles, 256^3 grid cells)
$t^{push} \simeq 115$ ($t_{tot} \simeq 116$) nsecs/particle/time step for S2(162 million particles, 128^3 grid cells).

To analyse the particle push stage of the code, in Fig. 8 wc show times spent in the various portions of particle push for S2 on the log scale. Since each processor has approximately equal number of particles in the scaled size problems, the times spent by "productive" particle compu- tations, *particle move* and current *deposit,* arc independent of the number of processors used. For practical applications, *particle move* and *current deposit,* are the most computational intensive portions of the code, For the S2(512node) case, $T_{move}/T^{push} \simeq 2.22(T_{current}/T^{push}) \simeq$ *0.67,* As for the communication times T_{trade} and T_{gcsm}, $T_{trade} > T_{gcsm}$ ($T_{trade} \simeq 10.4T_{gcsm}$ at $N_p = 512$) because particle trade has a loop over all the particles. Due to the large problem size on each processor, T_{trade} and T_{gcsm} arc negligible within T^{push} (At $N_p = 512$: $T_{trade}/T_{move} \simeq 0.04$, $T_{gcsm}/T_{current} \simeq 0.01$, and ($T_{trade} + T_{gcsm})/T^{push} \simeq 0.03$). Hence, although T_{trade} and T_{gdsm} increases somewhat as the node number increases, this increase of communication dots not affect T^{push}.

To analyst the field solve stage of the code, in Fig. 9 wc compare the field solve times for S1 and S2. The time spent to update the field $T_{fldupdate}$ is independent of N_p because scaled problems have equal number of grid points per processor. However, unlike the particle push, wc find the communication time T_{gcfl} is larger than $T_{fldupdate}$ for both S1 and S2 when $N_p > 1$. (At 512 nodes, $T_{gcfl}/T_{fldupdate} \simeq 4.6$ for S1 and $T_{gcfl}/T_{fldupdate} \simeq 1.8$ for S2,) This is because the finite-diflcrcnce field update is very fast and the number of grid points within each processor is not large enough. Since *Field Update* operates on grid points within a subdomain while *Guard Cell Exchange* operates on grid points over a subdomain surface, the ratio of $T_{gcfl}/T_{fldupdate}$ correlates with *S/V.* Through other test runs, wc find that one usually needs *S/V* $\leq$ *0.15* to achieve $T_{gcfl}/T_{fldupdate} < 1$. For the test runs used here, wc only have *S/V* = 0.1875 for S1 and *S/V* = 0.375 for S2. Nevertheless, since the code time is dominated by T^{push} and T^{field} is negligible in T_{tot}, T_{gcfl} only has a minimum effect on the overall code performance.

The timing results in Figs. 8 and 9 show that the guard cell communication times (T_{gcsm} and T_{gcfl}) increase as the number of processors increases. However, the guard cell number and the size and number of communicated message is determined only by the *S/V* ratio and, thus, independent of the number of processors for scaled problems. Hence, the observed increase of T_{gcsm} and T_{gcfl} is apparently a result of the Delta communication network contention.

3.2 Fixed Problem Size Analysis

in a fixed problem size analysis, we compare the times to run the same problem on an increasing number of processors. Since the total problem size is fixed, the problem size on each individual processor decreases as the number of processors N_p increases. For a problem that can be fit into a minimum of N_{min} processors, the parallel efficiency for $N \geq N_{min}$ processors is defined by

$$\epsilon(N) = \frac{T_{tot}(N_{min})N_{min}}{T_{tot}(N)N} \tag{18}$$

We consider the following two fixed size problems. The size of the first problem, F1, is $3'2^3 = 32768$ grid cells and 2.22×10^5 particles ($\sim$ 7 particle/c.ell). In the second problem, F2, we use the same number of grid cells but increase the total number of particles to 2.52×10^6 ($\sim$ 77 particle/cell). Note the size of F1 is the same as that of S1(1node), which can be fit on on a single processor on Delta. The size of F2 is the same as that of S2(8node), which requires a minimum of $N_{min} = 8$ processors to run. F1 and F2 were run using processors from $N_p = N_{min}$ to $N_p = 512$. 3D domain partitions arc used to decompose the total domain into A'_p subdomains of equal size.

In Figure lOa, wc plot the parallel efficiencies for F1 and F2 (left axis) and the ratio of T^{cbc}/T_{tot} (right axis, note scale change) as a function of processor number N_p. The loop time T_{tot}, particle push time T^{push}, and field solve time T^{field} are shown in Figure 10b. The results show that the efficiency decreases as the number of processors increases. This is to be expected because, for fixed size problems, the increase of N_p reduces the computations on each processor while increases interprocessor communications. As shown in Figure 10, the increase of communication overhead greatly affects the performance for small size problems such as F1. For instance, when we divide F1 in a 3D partition using $8 \times 8 \times 8 = 51?$ processors, each processor will only have a computation domain of 4^3 grid points and about 430 particles. With such a small size problem on each node, the communication time becomes larger than the computation time, which causes a low parallel efficiency of $\epsilon(512) \sim 27\%$. On the other hand, we find that F2 performs much better than F1 on multiple nodes due to its larger problem size. When F2 is divided into 512 processors, each processor still has on average $\sim$ 4922 particles, which is sufficient for computation time to dominate T_{tot} and for the parallel efficiency to reach $\epsilon(512) \simeq 76\%$. As expected, for PIC simulations, a parallel computer is best suited only for problems with a large enough size (i.e. total number of particles).

To examine in detail the communication costs for fixed size problems, in Fig. 11 we show T_{trade}, T^{gc}, and T^{cbc} normalized by T_{tot} as a function of N_p (left axis). (Recall that T_{trade} is the communication and boundary condition time associated with particles and T^{gc} is the communication and boundary condition time associated with field and current guard cells,) For fixed size problems, increasing the processor number increases the subdomain surface to volume ratio *S/V*, Since guard cell communication directly correlates with S/V and particle communication is also somewhat related to *S/V*, in Fig. 11 wc have also plotted *S/V* as a function of N_p (right axis). For F1 (Fig. 11a) which has $\sim$ 7 particles per cell on average, the guard cell communication time T^{gc} dominates the total communication time T^{cbc}. On the other hand, for F2 (Fig. 11b) which has $\sim$ 77 particles per cell, we have $T_{trade} \geq T^{gc}$. Due to the larger number of particles used in F2, T^{cbc}/T_{tot} is much less sensitive to the number of processors used and to the surface to volume ratio *S/V*. Therefore, to achieve a high parallel efficiency for PIC codes, the key parameter one can adjust is to make sure there are enough particles per processor.

Another factor that affects the particle communication time T_{trade} is the number of particles traded between processors. in Fig. 12, we plot T_{trade}/T_{tot} as a function of the percentage of particles traded per time step. For this analysis, we use F2(64node) and F2(256node) as an example. Comparing to F1 and the scaled size problems (S1 and S2), the F2 cases are the problems that need to trade more particles duc to their large *S/V* and large number of particles per cell. To get an even higher percentage of particles traded per time step, wc increase the time step dt. in Fig. 12, the number of particles traded in F2(64node) is increased from 1.7% to 11.4% and that in F2(256node) is increased from 3.2’%0 to 20.5% as dt is increased. While the percentage of particles traded is increased about 7 times, we find T_{trade}/T_{tot} stay as almost constant, This is because T_{trade} is mainly spent on the loop over particles to check particle positions, which is not affected by the number of particles traded. Hence, the performance of the code will not degrade for problems that require a large fraction of particles to be traded. Large fraction of particles may be traded in problems involving nonuniform particle distributions where small subdomains must be used for load balance.,

3.3 Performance on Delta vs. Performance on Cray

Finally, we compare the overall performance of the code on Delta with that on single processor Cray supercomputers. Two Cray computers were used for this analysis. The first one is the Cray Y-MP at JPL which has a memory limit of 16 Mwords or 128 Mbytes. The second one is the Cray C90 at NASA Ames (The Von Neuman). The memory limit on the NASA Ames Cray C90 is 128 Mwords or 1.024 Gbytes. The memory limit on Intel Delta is about 6 times larger than that of a single CPU Cray C90.

Other than the message-passing and global communications, the Cray version of the code is identical to the parallel version. The Cray version of the code is compiled using the Cray Fortran compiling system's automatic vectorization and optimization. However, no re-writing was done to optimize and vectorize the gather/scatter for the Cray. Hence, the code performance on Cray is not the best performance one can get from a Cray. All the Cray runs were carried out on a single CPU.

In Figure 13 we plot the total run time T_{tot} for the S2 case as a function of the "problem size". The unit of the problem size is defined as the problem size on 1 node of the Delta computer. For S2, the size unit is 3.16 x 10^5 particles and 16^3 grid cells. Note that while the problem size on the Cray processor varies, the problem size per processor on the Delta is constant because the number of processors increases with problem size. Due to the memory limits on the Cray supercomputer, not all S2 problems can be run on the Cray. For instance, the largest S2 problem we ran on the Cray C90 is $size \simeq 91.13$ (72^3 grid cells and 2.9 x 10^7 particles),

To compare the performance, we define the Delta speedup as

$$Speedup = \frac{(T_{tot}/size)_{Cray}}{(T_{tot}/size)_{Delta}} \tag{19}$$

For small problems, the Cray supercomputer performs much better than the parallel computer. Comparing to Cray C90, the speedup factor at $size = 1$ is $Speedup \simeq$ 0,12 for S2. However, as the problem size increases, the time spent on the Cray increases approximately linearly in the log scale. While on the Delta, due to the high parallel efficiency, the total run time for S2 stays almost constant as both the problem size and processor number are increased. At $size =$ 64, we find the speedup of the Delta over the Cray C90 is $Speedup \simeq 7.42$ for S2. Extrapolating the run times on Cray to $size =$ 512, if one had a Cray C90 or a Cray Y-MP large enough to run the $size =$ 512 problems, then the speedup of the Delta over the Cray C90 would be $Speedup \simeq 58.4$

and the speedup over the Cray Y-MP would be $Speedup \simeq 116.6$. We have also performed the same analysis for the S1 case, and obtained similar results.

4 Summary and Conclusions

A MIMI I parallel 3D electromagnetic PIC code has been developed on the 512 node Intel Touchstone Delta system. in the code, the particles are pushed using a standard relativistic leapfrog scheme and the electromagnetic field is updated locally using a rigorous charge-conservation finite-difference method. The code is implemented using the General Concurrent PIC (GCPIC) algorithm[3] which uses a domain decomposition to divide the computation among the processors. Three major message-passjng operations, *particle trade*, *guard cell exchange,* and *guard cell summation,* are used to link the computations in different processors together. With 12 Mbytes memory per node and a total of about 6 Gbytes on all 512 nodes available to users, the Intel Delta system allows our code to run simulations using over 108 particles and 10^6 grid cells. The parallel efficiency of this code was evaluated using both fixed problem analysis and scaled problem analysis. It is shown that our 3D EM PIC code runs with a high parallel efficiency of $c \geq 95\%$ for large size problems. The particle push time we have achieved is 115 nsccs/particle/time step for 162 million particles on 512 nodes. The overall performance of the code on the Delta is also compared with that on Cray supercomputers. Comparing with the runs on a Cray C90, we have observed a factor of 58 speedup on the Delta for our test runs.

We find, for parallel computers, a finite difference field solve is significantly more efficient than fast Fourier transforms. Our result shows that the finite difference field solve takes $< 0.7\%$ of the total CPU time for problems with about ~ 77 particles/cell and $\leq 3\%$ for problems with ~ 7 part icles/cell. 'J'he field solve time is ≤ 4 % even for problems ~ 5 particles/cell. This implies that the effect of load balance for the field solve is not nearly as important as that for the particle push, Hence, when the code is applied to problems with nonuniform particle distributions, a high parallel efficiency can still be achieved as long as the domain is decomposed according to particle load balance.

As expected, the major factor that can degrade the code's performance is the number of particles per processor. However, we find the "bandwidth" on number of particles/processor is fairly large for high efficiency simulations. For instance, when using 256 to 512 processors, our test runs have a parallel efficiency $\epsilon \geq 95\%$ for $\geq 2 \times 10^5$ particles/processor, $\epsilon \sim 86\%$ for $\sim 2 \times 10^4$ particles/processor, and $\epsilon \sim 76\%$ even for small problems with only $\sim 5 \times 10^3$ particles/processor.

Acknowledgments

We would like to thank E. Huang (JPL) and P. Lyster (JPL) for their help in implementing the code on the Intel Delta and for many useful suggestions throughout this research. We would also like to thank F. Tsung (UCLA) for many useful discussions, E. Mattson (JPL) for helping to produce the color figures (Figs. 3 and 5), and B. Mackey (JPL) for assistance in simulations using the Cray C90. This work was carried out by the Jet Propulsion Laboratory under contracts from Sandia National laboratory and US Department of Energy through agreements with NASA. Access to the Intel Touchstone Delta System, which is operated by Caltech on behalf of the Concurrent Supercomputing Consortium, was provided by NASA. Access to the Cray supercomputers was provided by the JPL Supercomputing Project, which is sponsored by JPL and NASA Office of Space Science and Applications,

References

[1] C.K. Birdsall and A.B. Langdon, Plasma Physics via Computer Simulation (McGraw-Hill, New York, 1985).

[2] R.W. Hockney and J.W. Eastwood, Computer Simulation Using Particles (McGraw-Hill, New York, 1981).

[3] P. C. Liewer and V. K. Decyk, A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes, *J. Computational Physics*, 85 (1989), 302-322.

[4] P. C. Liewer, V. K. Decyk, J. M. Dawson, and B. Lembege, Numerical Studies of Electron Dynamics in Oblique Quasi-Perpendicular Collisionless Shock Waves, *J. Geophysical Research*, 96 (1991), 9455-9465.

[5] T. J. Krucken, P. C. Liewer, R. D. Ferraro, and V. K. Decyk, A 2D Electromagnetic PIC Code for Distributed Memory Parallel Computers, in *Proceedings of the 6th Distributed Memory Computing Conference*, (IEEE Computer Society Press, Los Alamitos, CA), (1991), p.452.

[6] D.W. Walker, Particle-in-Cell Plasma Simulation, Codes on the Connection Machine, *Computing Systems in Engineering*, 2 (1991), 307-319.

[7] N. G. Azari and S.-Y. Lee, Hybrid Task Partitioning for Particle-in-Cell Simulation on Shared Memory Systems, *Proceedings of Intl. Conf. on Distributed Computing Systems, Dallas, TX,*, May 1991, p. 526.

[8] S.-Y. Lee and N. G. Azari, Hybrid Task Decomposition for Particle-in-Cell Methods on Message Passing Systems, Proceed ings of International Congerence on Parallel Processing, St. Charles, IL, August 1992, Vol. 111, p. 141.

[9] P.M. Lyster, P. C. Liewer, V. K. Decyk, and R. D. Ferraro, Implementation and Characterization of a 3-D Particle-in-Cell Code on MIMD Massively Parallel Supercomputers, (submitted for publication, 1994).

[10] M. Kiefer, D.B. Seidel, R.S. Coats, J.P. Quintenz, T.D. Pointon, and W.A. Johnson, Architecture and Computing Philosophy of QUICKSILVER, Proceedings of Conference on Codes and the Linear Accelerator Community (1990).

[1 1] QUICKSILVER Programmer's Guide, Sandia National laboratories.

[12] J. Villasenor and O. Buneman, Rigorous Charge Conservation for Local Electromagnetic Field Solvers, *Computer Physics Communications*, 69 (1 992), 306-316.

[13] O. Buneman, T. Neubert, and K-1 Nishikawa, Solar Wind-Magnetosphere Interaction as Simulated by a 3-D EM Particle Code, *IEEE Trans. Plasma Science, 20 (1992), 810-816.*

14] K. S. Yet, Numerical Solution of initial Boundary Value Problems involving Maxwell's Equations in Isotropic Media, *IEEE Trans. Antennas Propagat., 14* (1 966), 302-307.

15] R. D. Ferraro, P. C. Liewer and V. K. Decyk, Dynamic Load Balancing for a 2D Concurrent PIC Code, *J. Computational Physics*, 109 (1 994), 329-340.

[16] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol.1 (Prentice-Hall, New Jersey, 1988)

Figure Captions

Fig. 1. The Yee lattice showing the location of the field components on the staggered, finite-difference mesh, where grid points are at the corners of the cell shown. The electric field and current components are defined at the mid-points of cube edges, while the magnetic field components are defined at the centers of cube faces.

Fig. 2. Domain decomposition of a two-dimensional PIC simulation for 4 processors showing processor subdomains and guard cells (shaded). Each processor is responsible for updating all the particles and grid points in its' subdomain. Each processor also stores field information for the guard cells surrounding its subdomain to minimize interprocessor communication.

Fig. 3. Domain decompositions for 1-, 2- and 3-dimensional sub domains with particles colored by processor.
(a) 1-D partition for 4 processors. (b) 2-D partition for 16 processors; and (c) 3-D partition for 64 processors,

Fig. 4. Flow chart for the parallel 3D electromagnetic code showing the "particle push" and "field solve" stages. The routines in the rounded boxes require no interprocessor communication, while the routines in the five rectangular boxes involve interprocessor communication, Global boundary conditions are also imposed in these "guard cell" routines.

Fig. 5. Electron (x, z, v_x) phase space from the. test case used in the performance analysis, a relativistic electron two stream instability.
Top: Electrons at t=0 colored by processor showing the domain decomposition in the x-z plane.
Middle: Electrons at t=0 colored by populations showing two counter streaming electron beams (drifting velocity $v_d = \pm 0.4c$).
Bottom: Electrons at $t\omega_{pe} = 47$ colored by species showing the phase-space mixing of the beams caused by the instability.

Fig. 6. Code performance for two scaled problems where the problem size increases with the number of nodes, N_p. Per node, Case S1 has 114 million particles and 256^3 grid points and Case S2 has 162 million particles and 128^3 grid points. The parallel efficiency (left axis) is very high > 9570 even on 512 processors. The right axis shows T_{cbc}/T_{tot} (right axis) versus the number of processors where T_{cbc}, defined in eq. (]5), measures the communication overhead.

Fig. 7. Comparison of run times of different code portions for scaled problems S1 (a) and S2(b).

Shown arc the total loop time, T_{tot}; the particle push time, T^{push}; the particle communication time (trade, current guard ccl], and boundary conditions), $T_{trade} + T_{gdsm}$; the field solve time, T^{field}; and the field communication time (field guard cells), T_{gdfl}.

Fig. 8. Analysis of the parallel push stage for S2. Plotted are times spent in the four subroutines of the particle push stage: particle move T_{move}, particle trade T_{trade}, current deposit $T_{current}$, current guard cell + boundary conditions T_{gdsm}, and the total particle push time T^{push}. The particle move and deposit dominate the computation.

Fig. 9 Analysis of the field solve stage for scaled problems S1 and S2. Time spent in portions of the field solve: field update $T_{fldupdate}$ and field guard cell + boundary conditions, T_{gcsm}. For both cases, the time spent in interprocessor communication dominates, but the total cost of the field solve is so low that the total code efficiency remains high.

Fig. 10. Code performance for fixed-sized problems F1 and F2 vs. the number of processors N_p. The problem size pcr node decreases as the number of processors increases; increasing the interprocessor communication.
(a) Parallel efficiency (open symbols, left axis) and the communication time T^{cbc}/T_{tot} (right axis, filled symbols). (b) Total time T_{tot}, time for push stage T^{push}, and time for field solve stage T^{field} versus N_p. The push stage dominates for both cases.

Fig. 11, Analysis of the communication/boundary condition times (left axis) $T^{gc} = T_{gcsm} + T_{gcfl}$, T_{trade}, and $T^{cbc} = T_{gc} + T_{trade}$ for fixed size cases F1(a) and F2 (b) vs. the number of processors N_p. In (a), for F1($\sim$ 7 particles/cell), the guard cell time T^{gc} exceeds the particle trade time, whereas in (b) for F2($\sim$ 77 particle.s per cell), the times arc comparable. Also shown is the subdomain surface to volume ration *S/V*. Both S/V and the communication costs increase as the size of the subdomain shrinks with increasing N_p.

Fig. 12. Ratio of time spent in particle trade routine. (T_{trade}) to total loop time T_{tot} as a function of the percentage of particles traded per processor per time step. The timings arc for the F2 case, which trades the most particles comparing to the other cases. The ratio is < 10% and is insensitive to the actual percentage traded because most time is spend in the loop which checks all the particles to determine which need to be traded.

Fig. 13. Comparison of the Delta and single processor Cray YMP and Cray 90 performance versus problem size. For both Crays, the run time increases with the problem size since only one

processor was used. For the Delta, the number of nodes was increased with the problem size so the problem size per processor remained constant (Case S2). The sequential Cray code was not optimized for vectorization other than complied using the Cray systems automatic vectorization.

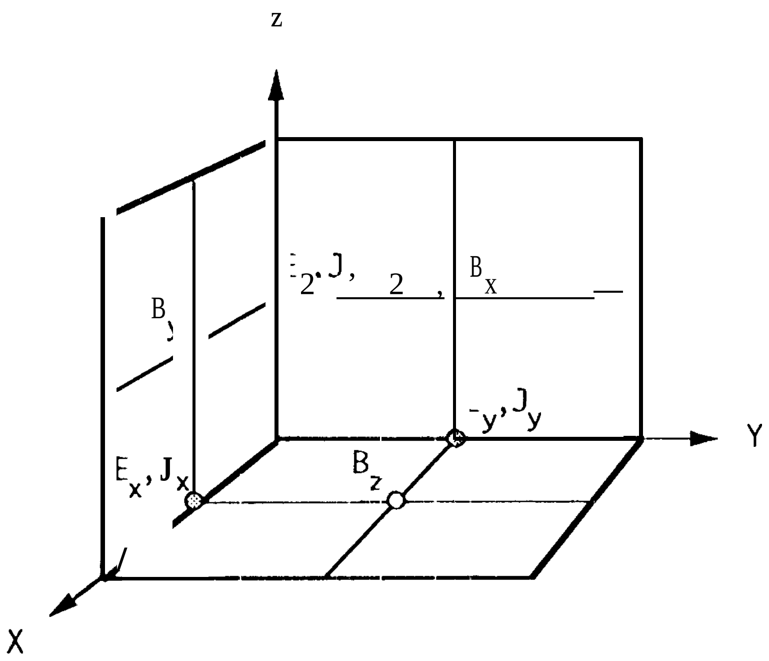


Figure 1

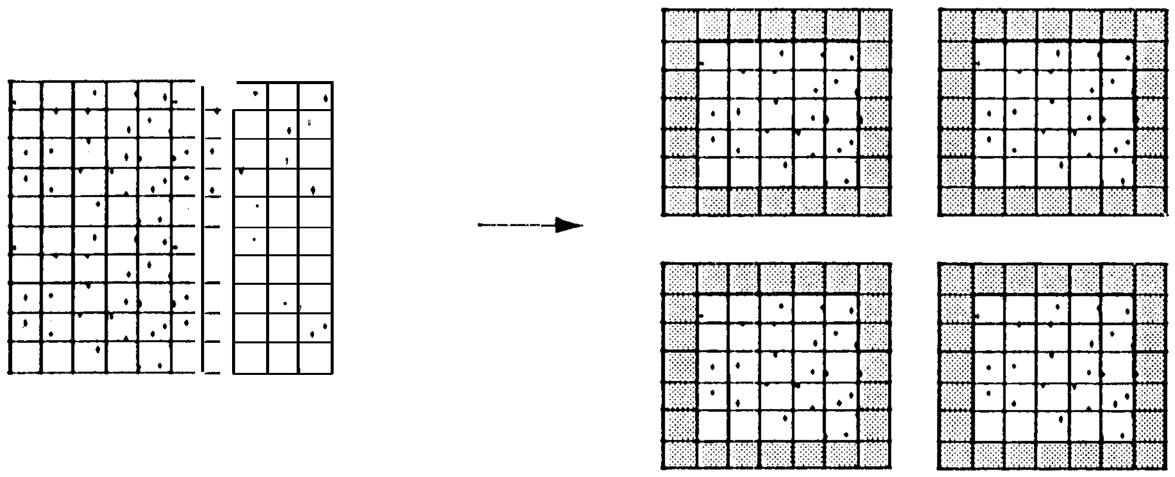


Figure '2

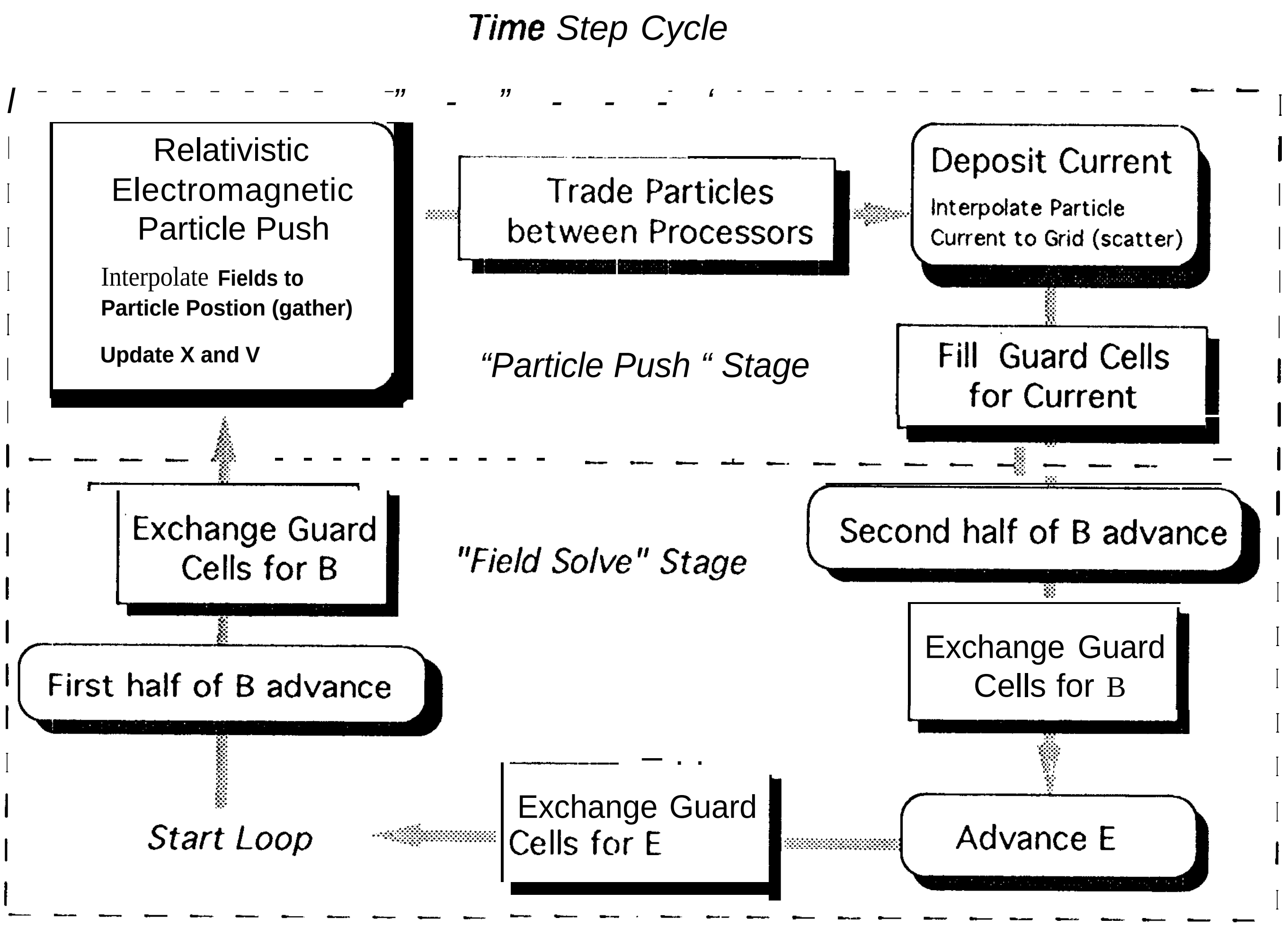
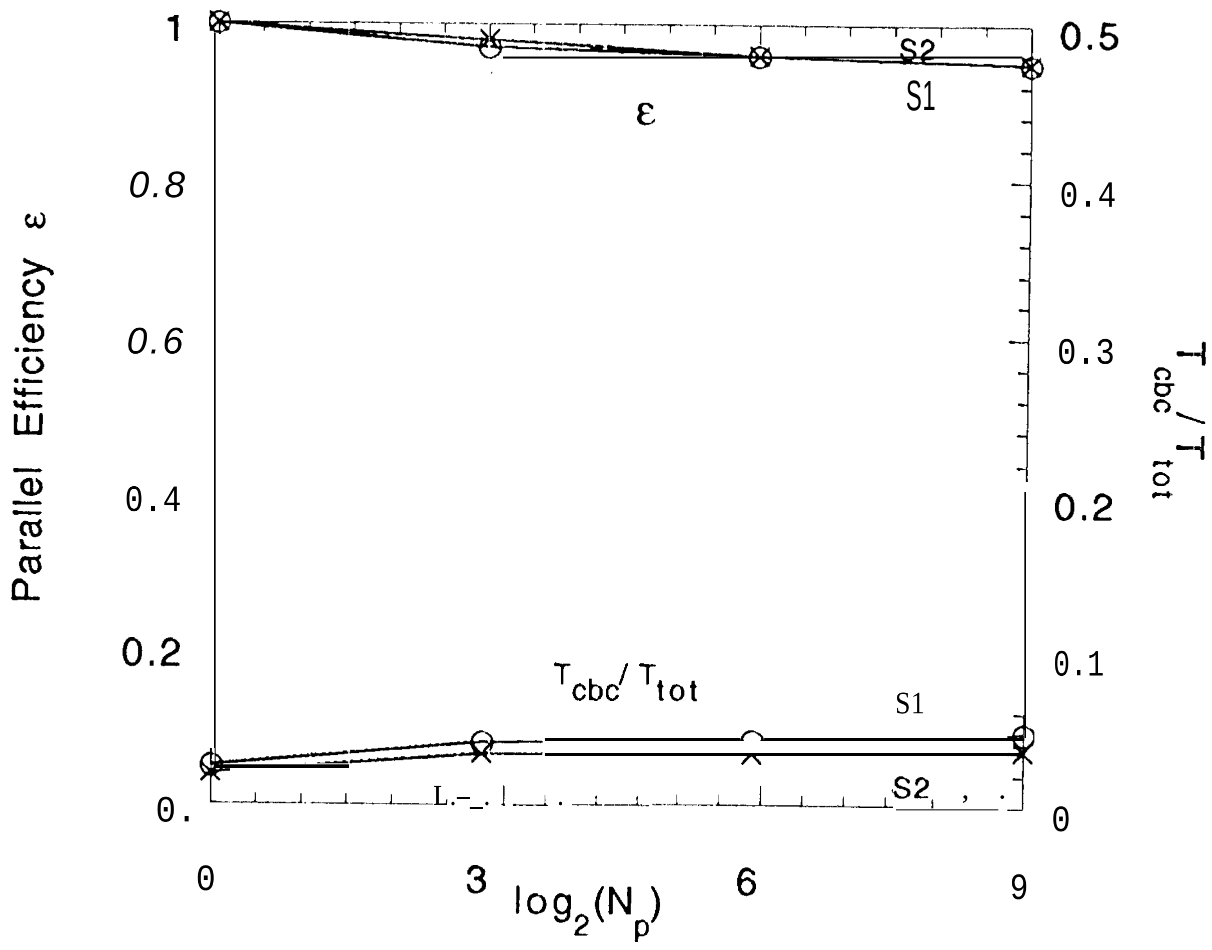
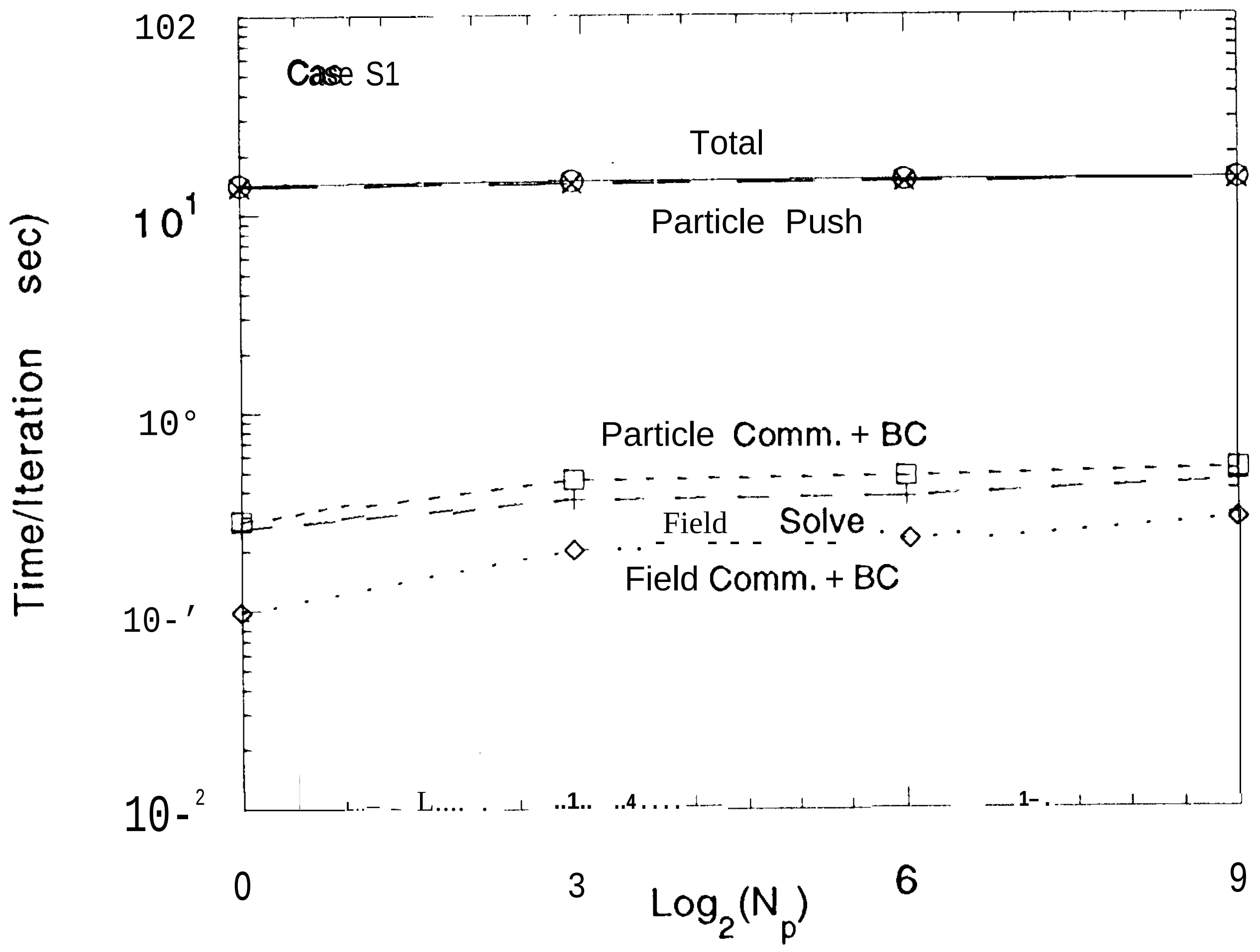

Time Step Cycle
Relativistic Electromagnetic Particle Push
Interpolate Fields to Particle Postion (gather)
Update X and V
Trade Particles between Processors
Deposit Current
Interpolate Particle Current to Grid (scatter)
Fill Guard Cells for Current
"Particle Push " Stage
"Field Solve" Stage
Second half of B advance
Exchange Guard Cells for B
Exchange Guard Cells for B
First half of B advance
Start Loop
Exchange Guard Cells for E
Advance E

Figure 4



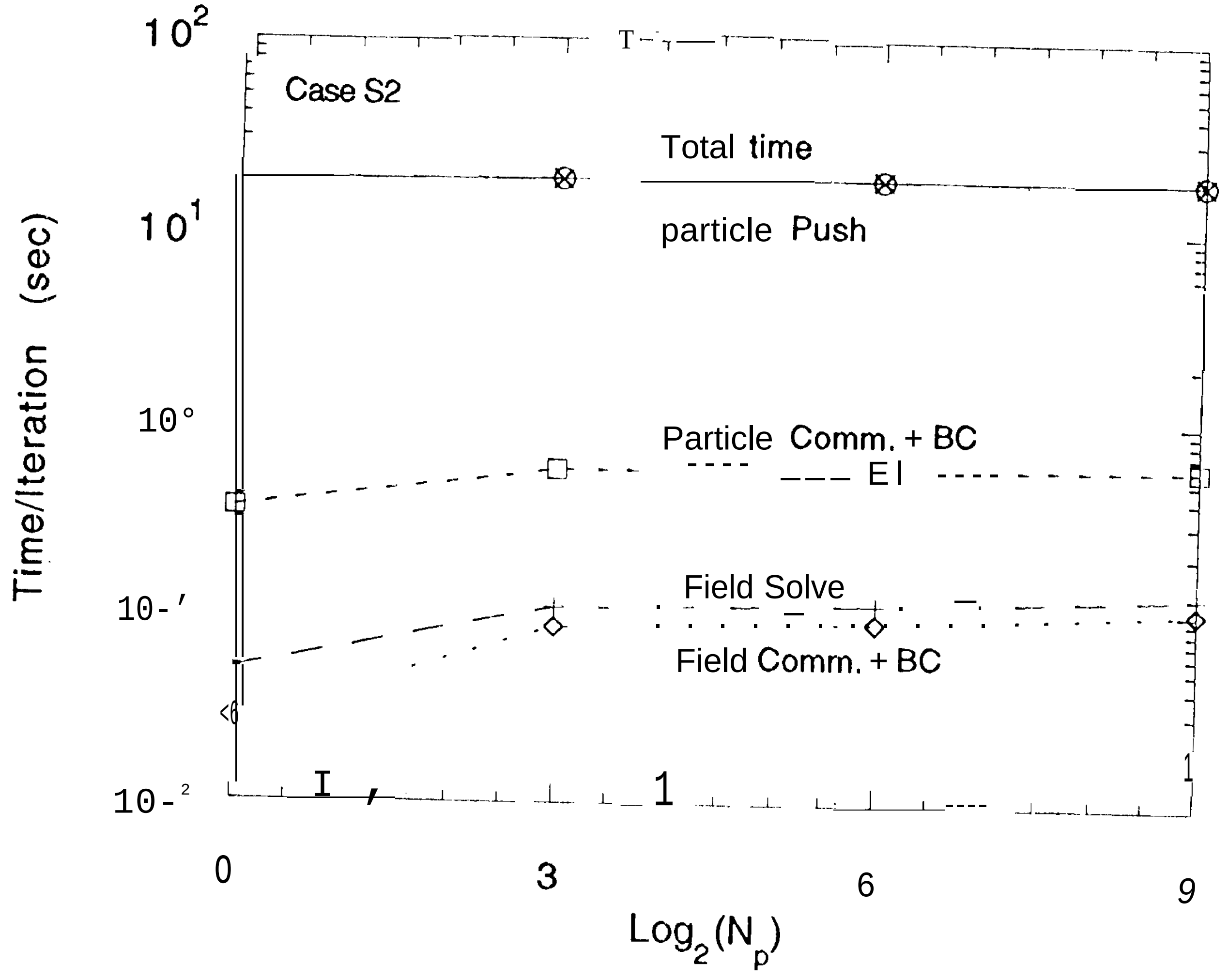
Parallel Efficiency ε
1
0.8
0.6
0.4
0.2
0.
0.5
0.4
0.3
0.2
0.1
0
T_{cbc} / T_{tot}
S2
S1
ε
T_{cbc}/ T_{tot}
S1
S2
0
3
6
9
$\log_2(N_p)$

Figure 6

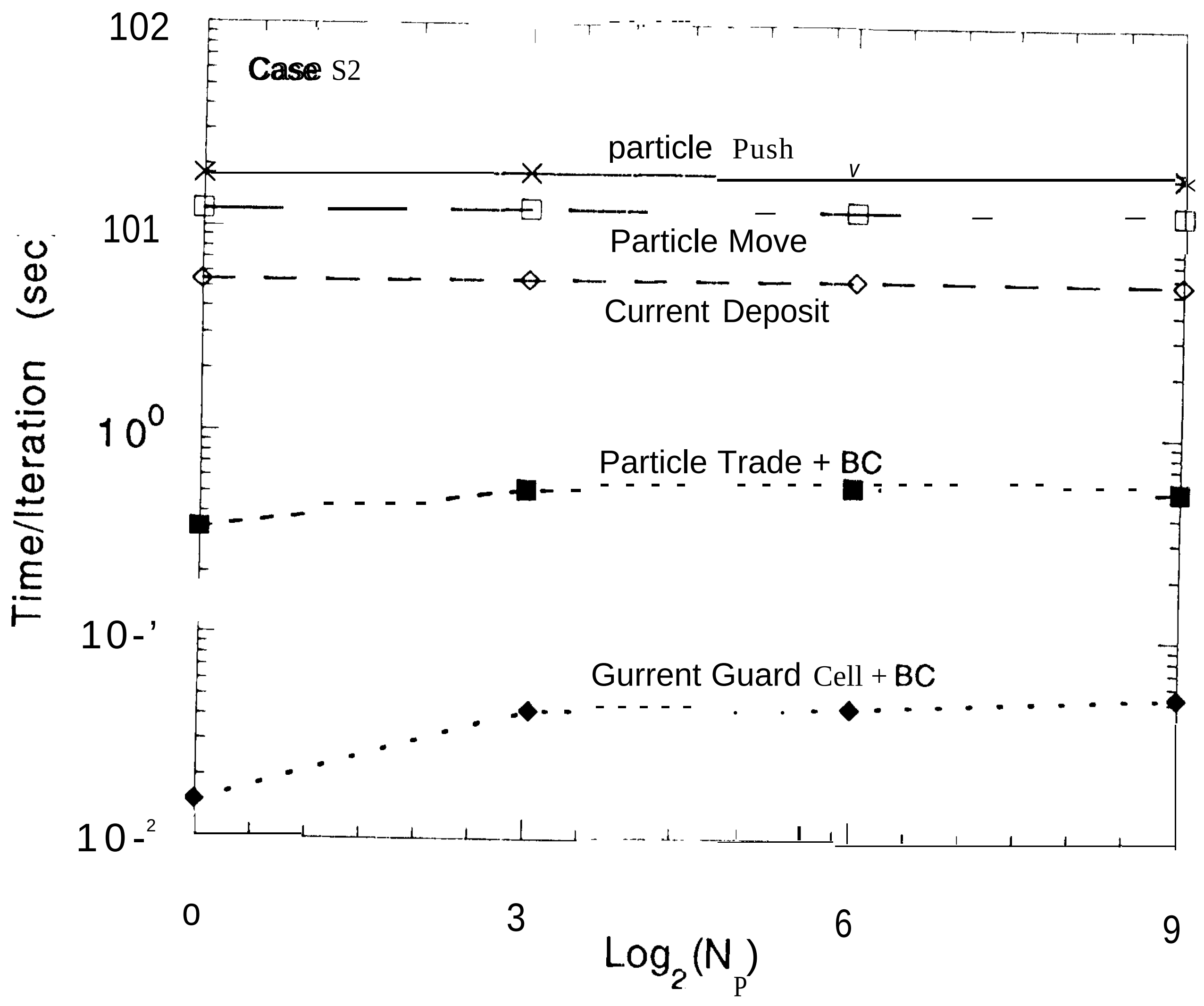


Case S1
Total
Particle Push
Particle Comm. + BC
Field Solve
Field Comm. + BC
102
10¹
10°
10-′
10-²
Time/Iteration sec)
0
3
6
9
Log2(Np)

Figure 7a

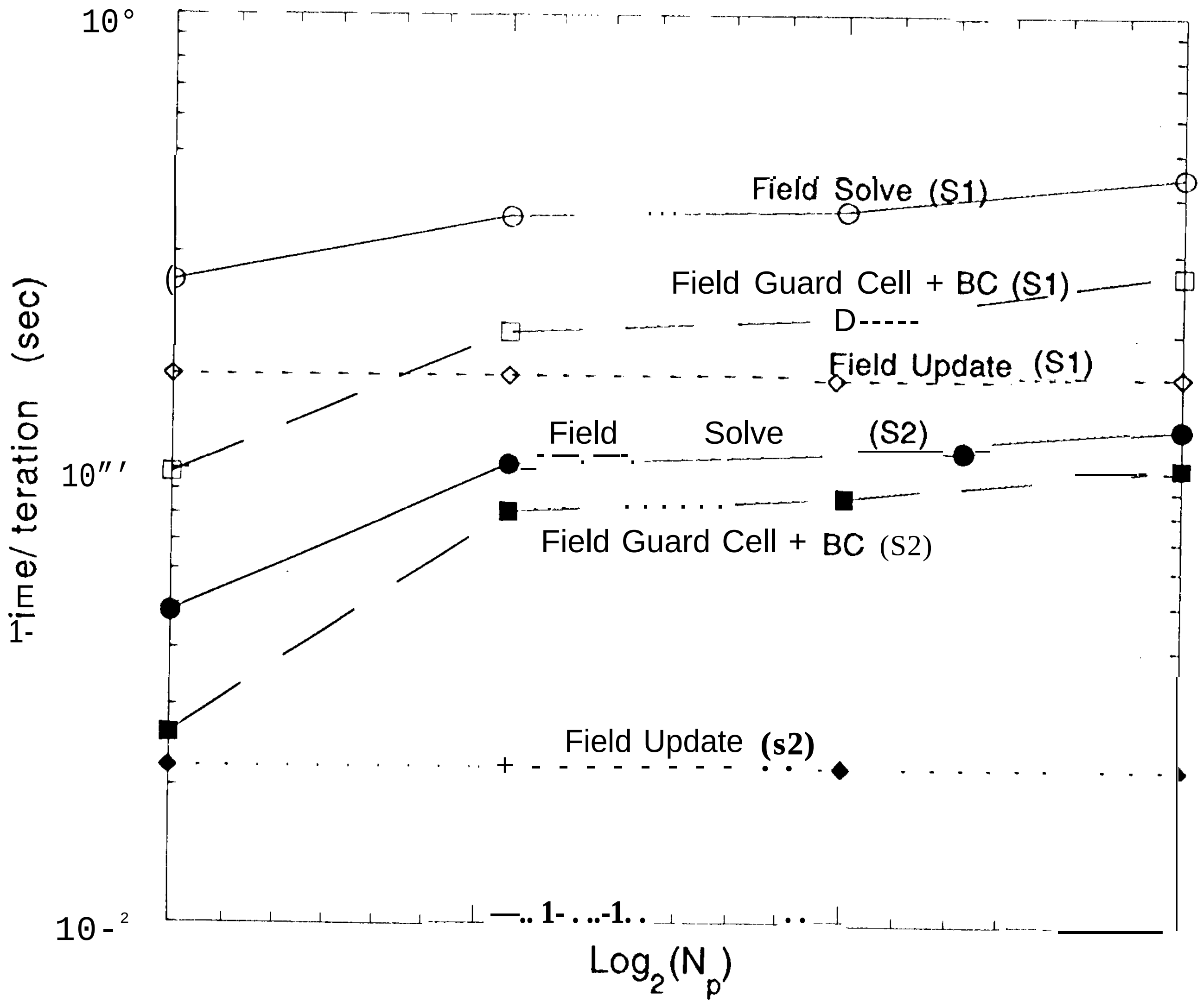

Case S2
Total time
particle Push
Particle Comm. + BC
Field Solve
Field Comm. + BC
Time/Iteration (sec)
10^2
10^1
10^0
10^{-1}
10^{-2}
0
3
6
9
$\mathrm{Log}_2(N_p)$

Figu e 7b



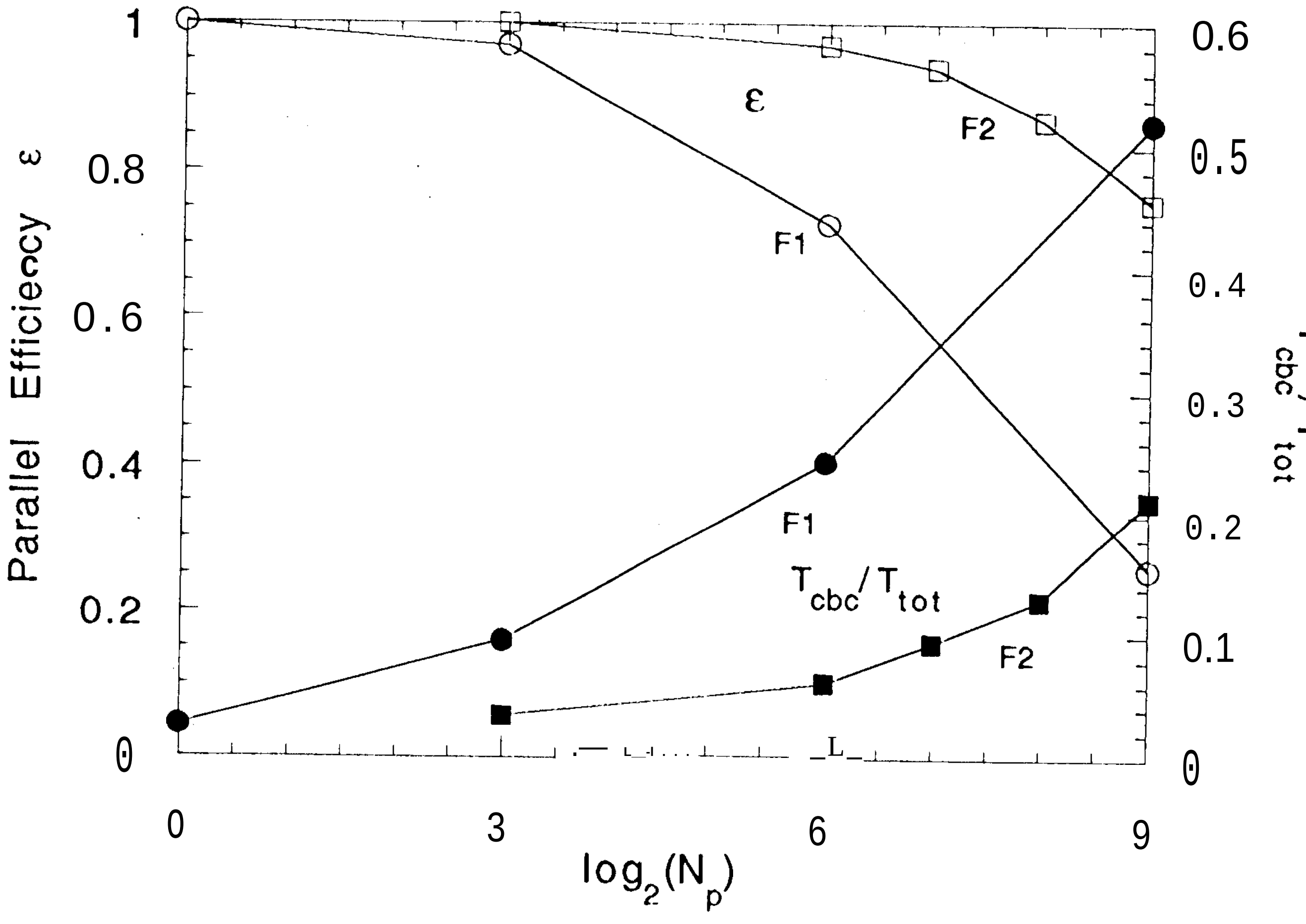
102
101
10-'
10-2
Time/Iteration (sec
Case S2
particle Push
Particle Move
Current Deposit
Particle Trade + BC
Gurrent Guard Cell + BC
0
3
6
9
Log2(Np)

Figure 8



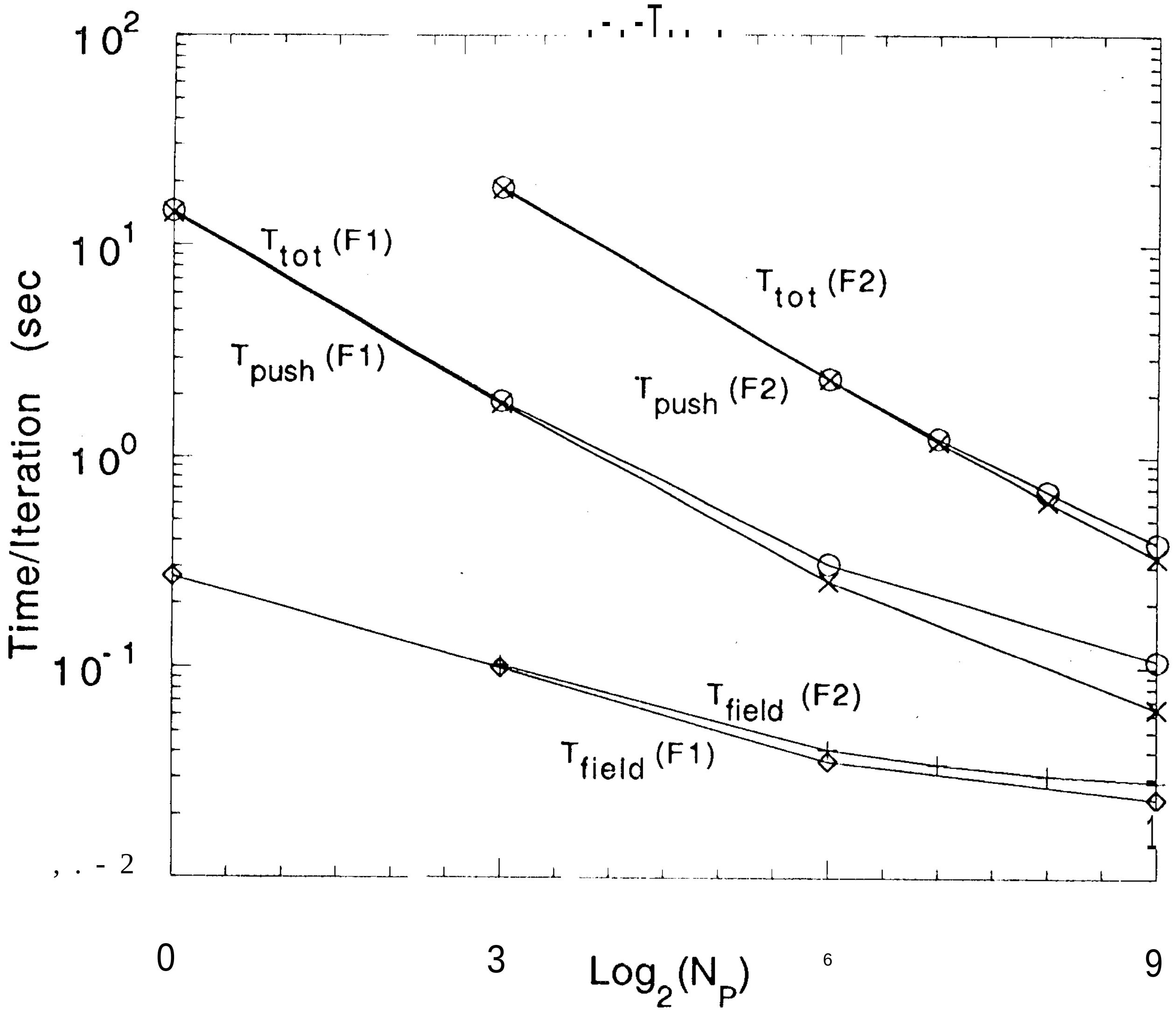
10°
10″′
10 -²
Time/ teration (sec)
Field Solve (S1)
Field Guard Cell + BC (S1)
Field Update (S1)
Field Solve (S2)
Field Guard Cell + BC (S2)
Field Update (s2)
Log$_2$(N$_p$)

Figure 9



Parallel Efficiecy ε
T_cbc/T_tot
log_2(N_p)
ε
F1
F2
F1
F2
T_cbc/T_tot
0
0.2
0.4
0.6
0.8
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0
3
6
9

Figure 10a



10^2
10^1
10^0
10^{-1}
10^{-2}
Time/Iteration (sec
T_{tot} (F1)
T_{push} (F1)
T_{tot} (F2)
T_{push} (F2)
T_{field} (F2)
T_{field} (F1)
0
3
6
9
$\mathrm{Log}_2(N_P)$

Figure 10b

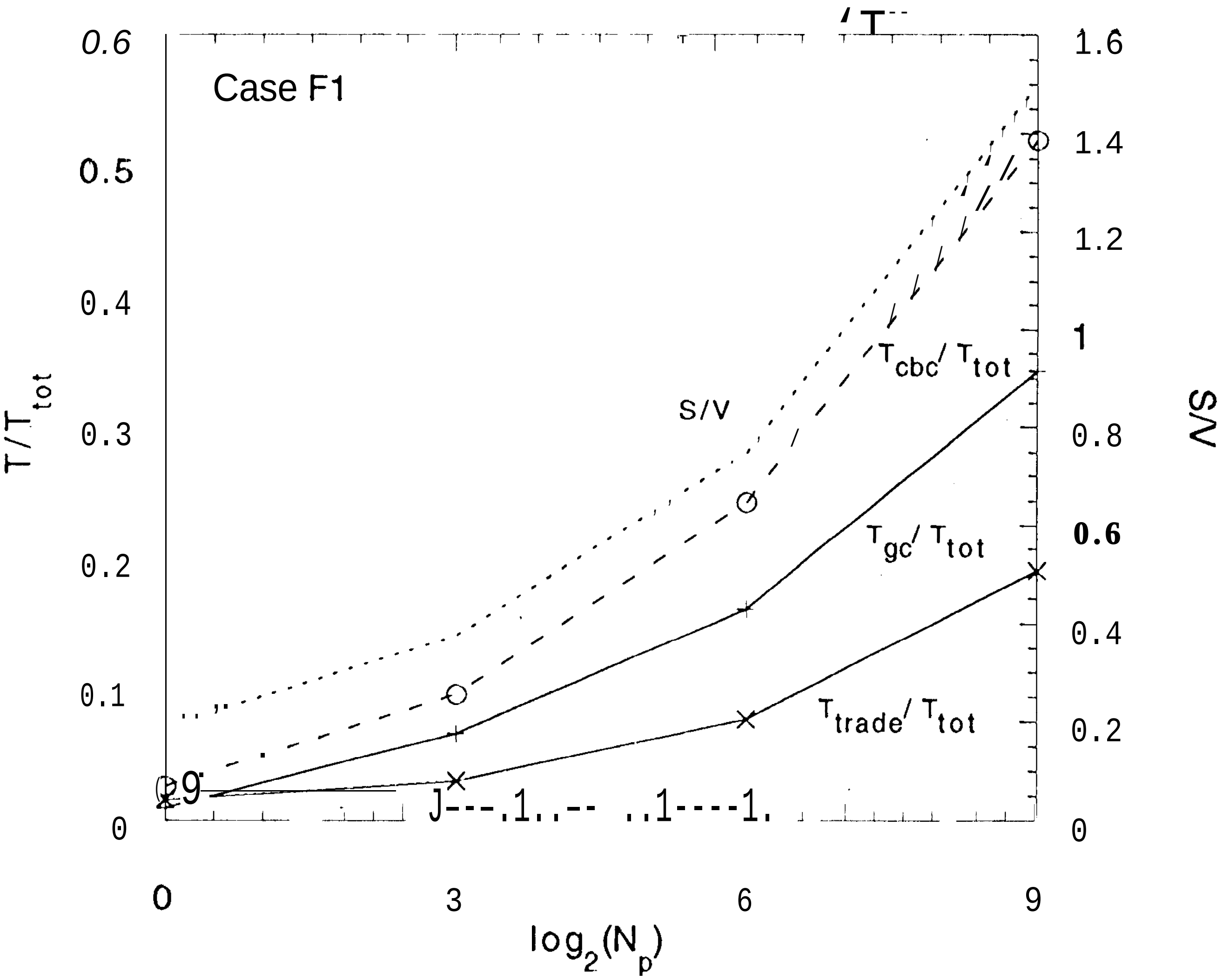
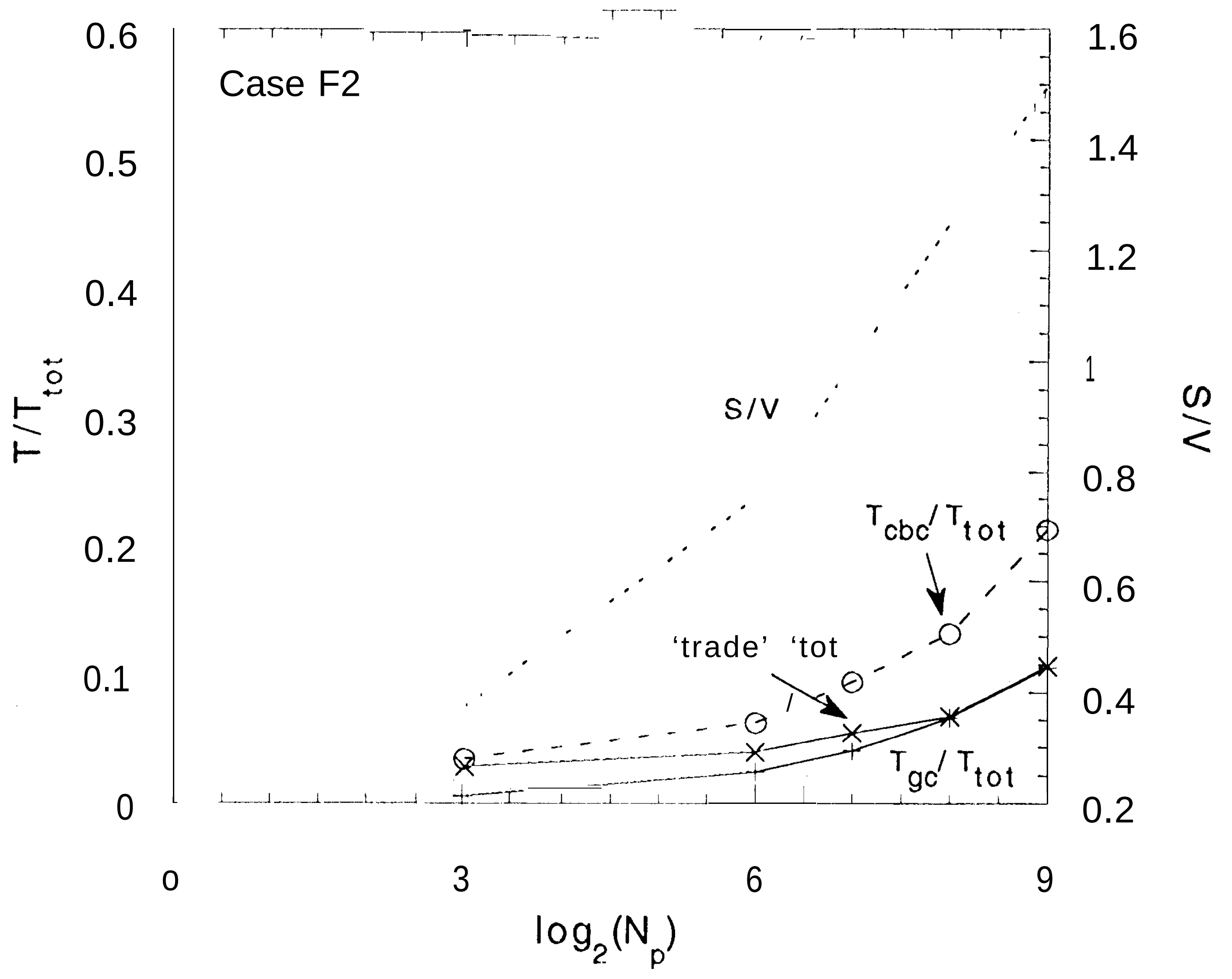

Case F1
T/T_tot
S/V
T_{cbc}/T_{tot}
T_{gc}/T_{tot}
T_{trade}/T_{tot}
log_2(N_p)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
0
3
6
9

Figure 11 a



Case F2
T/T_tot
S/V
S/V
T_cbc/T_tot
'trade' 'tot
T_gc/T_tot
log_2(N_p)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
o
3
6
9

Figure 1 lb

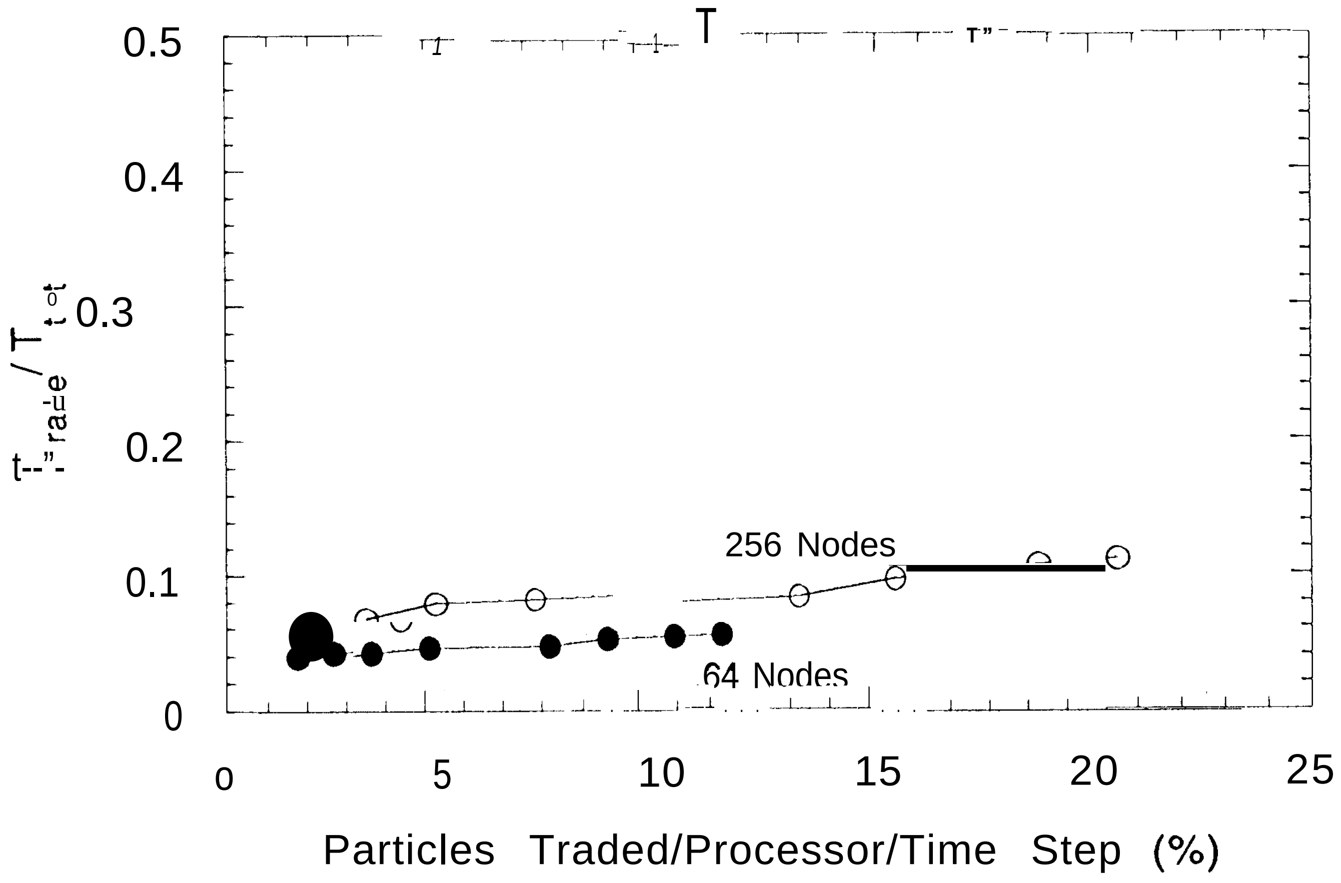
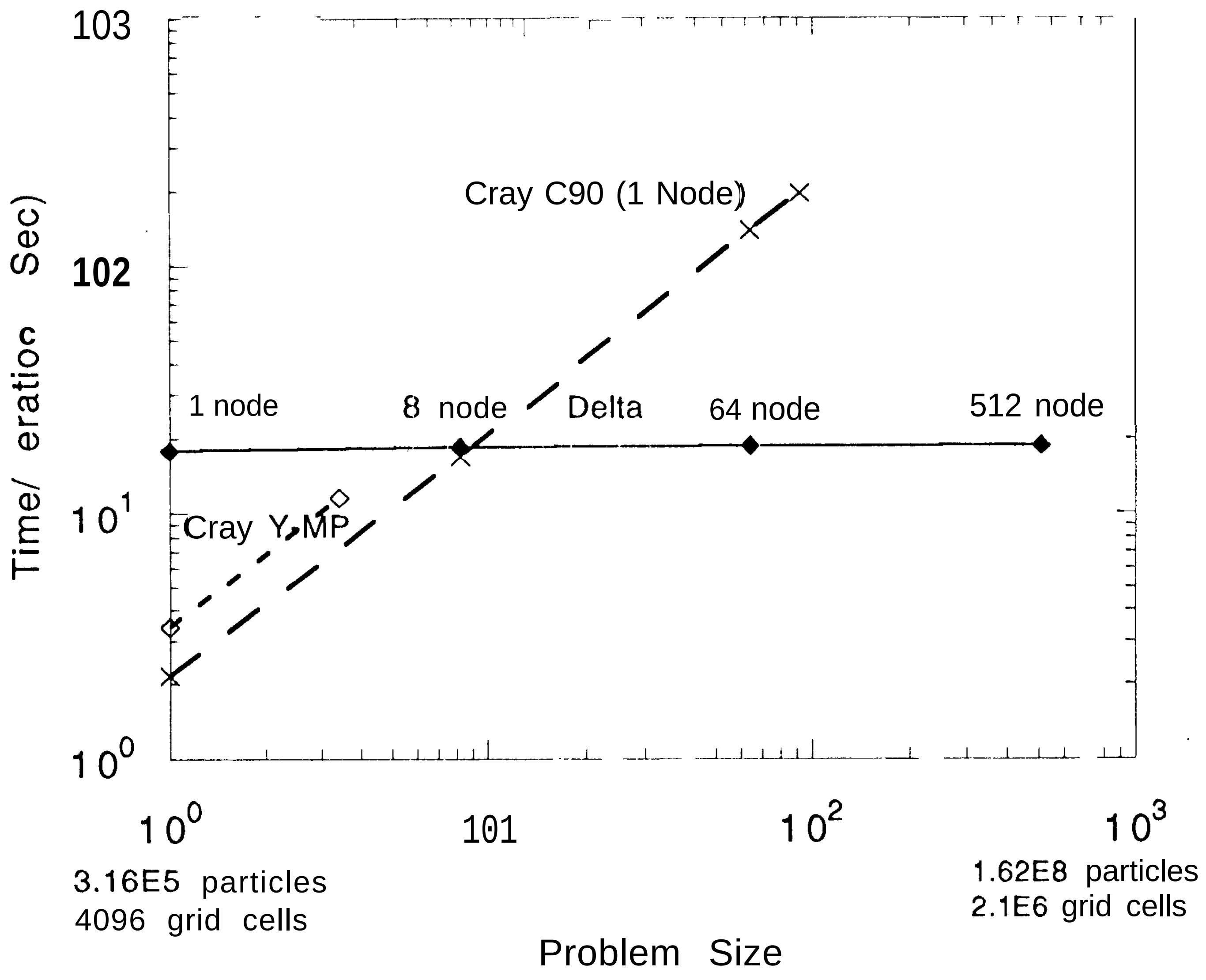

0.5
0.4
0.3
0.2
0.1
0
t_{trade}/T_{tot}
256 Nodes
64 Nodes
0
5
10
15
20
25
Particles Traded/Processor/Time Step (%)

Figure 12



103
102
10^1
10^0
Time/ eratioc Sec)
Cray C90 (1 Node)
1 node
8 node
Delta
64 node
512 node
Cray Y-MP
10^0
101
10^2
10^3
3.16E5 particles
4096 grid cells
1.62E8 particles
2.1E6 grid cells
Problem Size

Figure 13